# ADVANCED ALGORITHMS PREVIOUS EXAMS

(answers for the 1st part from 2020 & 2021 exams + some extras)

Davide Giannubilo a.a. 22/23

## Describe the Karger's Min Cut Algorithm

It is a randomized algorithm to compute a min cut of a connected graph.

Given a graph G = (V, E), find a partition V1, V2 such that the number of edges between them is minimized.

- The contraction of an edge merges the nodes *u* and *v* into a new one *uv*, reducing the total number of nodes of the graph by one.
- All other edges connecting either *u* or *v* are "*reattached*" to the merged node, effectively producing a multigraph.

Karger's basic algorithm iteratively contracts randomly chosen edges until only two nodes remain; those nodes represent a cut in the original graph. By iterating this basic algorithm, enough times, a minimum cut can be found every output can lead to a different solution since it is based on a random function.

The complexity is $O(n^4 \log n)$ (randomized algorithm) with an error probability of 1/poly(n).

## Describe how Karger and Stein algorithm works

The idea is based on Karger's algorithm.

From a multigraph G, that has at least 6 vertices, repeat twice:

- Run the original algorithm down to $\frac{n}{\sqrt{2}} + 1$ vertices
- Recurse on the resulting graph

Return the minimum of the cuts found in the two recursive calls.

The choice of 6 will only affect the running time by a constant factor.

$$T(n) = 2n^2 + 2T\left(\frac{n}{\sqrt{2}}\right) = O(n^2 \log n)$$

Any graph has at most $O(n^2)$ minimum cats.

## Describe how randomized QuickSort works

Quicksort is recursive in-place algorithm. It picks an element as a pivot and the partitions the array around it.

Choose a *pivot* element, declare 2 indexes *i* and *j*: *i* start from the beginning of the array and *j* starts from the end of the array.
*i* moves right until finds an element equal or greater than the pivot, instead, *j* moves left until it finds an element equal or smaller than the pivot; at this point swap elements pointed by the indexes and continue. When *j* becomes smaller than *i*, swap the pivot and the element pointed by

*j*, in this way the pivot will be in the right position and all the elements on the left will be smaller and all the elements on the right will be greater than it.

Now order subarrays calling recursively Quicksort function and that's it.

Usually, we choose the first or the last element as pivot but in the randomized version we choose a random one.

Best case: when the partition process always picks the middle element as the pivot $\Theta(n \log n)$.
Worst case: array sorted in reverse order or partition around max or min element $\Theta(n^2)$
Average case: with the random pivot (even if the partition does not split well the array) $\Theta(n \log n)$

## Describe how Radix Sort and Counting Sort algorithms interact

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. To do this, radix sort uses *counting sort* as a subroutine to sort digit by digit.

Counting sort is a non-comparison algorithm, and it uses a temporary array, so it is a *non in place algorithm*.
It is a sorting technique that works by counting the number of objects having distinct key values and then does some arithmetic to calculate the position of each object in the output sequence.

Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted.

We have:

- *n* is the number of elements
- *b* is the base or bucket size used (int 32 bit)
- *r* is the number of bits of each number (8 bit)
- *d* = $\frac{b}{r}$ is the number of digits in the given list (numbers of 4 digits, e.g., 1234)
- *k* = $2^r$ is the range of the input

Counting sort complexity is $\Theta(n + k)$ = $\Theta(n + 2^r)$.

Radix sort complexity is $\Theta\big(d(n + k)\big)$ = $\Theta\left(\frac{b}{r}(n + 2^r)\right)$. Increasing r means fewer passes but the time grows exponentially. If we choose $r = \log n$, this implies that $T = \Theta(bn/\log n)$ and since $b = d \log n$, we have that radix sort runs in $\Theta(dn)$.

## Describe the randomized select algorithm (Rand Select)

Randomized select algorithm is a random-algorithm, based on random partition algorithm, that computes the *i-th* smallest values of an array. The algorithm works applying divide and conquer technique: it runs recursively the random-partition algorithm over the upper or lower part of a given array with respect to the pivot element.
The pivot element of random-partition algorithm is the element that has all the lower elements before itself and all the greater elements after itself (even if in both cases they could be not necessarily ordered).
Random-partition algorithm select the pivot element uniformly random from a given array,

avoiding worst case scenarios such as when the array is ordered in increasing or decreasing order. To decide which partition has to be considered at each pass, random select algorithm compare the *i-th* index with the index of the partition: in case the index given in input is lower than the position of the pivot, run the recursion on the lower part or vice versa.

The expected time is $\Theta(n)$ but in the worst-case is $\Theta(n^2)$.
To speed-up the worst-case scenario, a good pivot has to be chosen recursively by grouping the array in blocks of 5 elements and finding the median of each group; then create a group with the median elements and find again the median, choosing it as pivot element. Even if it has been proved that with this approach the worst-case performance is linear, the first version of the algorithm is more practical.

## Describe how treap split/union works

A treap is a randomized data structure that combines the properties of trees with the property of a (min-)heap; every element *e1* has a priority *p* and a key *k*. So, at any moment the tree is equivalent to a classic binary tree if the elements were inserted in the order of their priority.

Union: we have two treaps *T1* and *T2* and we need to determine a key *k* such that

$$\forall\, x1 \in T1, \forall\, x2 \in T2 : key(x1) < k < key(x2)$$

Then, generate an element *x* with *key(x)* = *k* and *prio(x)* = $-\infty$.
Generate a treap *T* with *x* as root, *T1* as left subtree and *T2* as right subtree.
Rotate down *x* until it becomes a leaf and then, delete it from the treap.

Split: we have a treap *T* and we want to split it into two parts, *T1* and *T2*.
Find a key *k* that

$$\forall\, x1 \in T1, \forall\, x2 \in T2 : key(x1) \leq k < key(x2)$$

If there exists an element, delete it from T and obtain two treaps, T1 and T2; then reinsert the element deleted into T1.
If there is no element that satisfies that property, create a new element *x* with *key(x)* = *k* and *prio(x)* = $-\infty$, insert it into T and delete the new root in order to obtain two subtrees, T1 and T2.

The expected running time for union and split is $O(\log n)$.

## Describe how skip list insertion/search works

The skip list is randomized list-like structure. It is a list with multiple levels and each node of *i* level has *i* pointers.

Search: starting from the top level, go right until the element pointed is bigger than the one to search; then go back to the previous element and go down of one level and then, continue in that way until the element pointed and the element searched are equal or when we reach the end of the list.

Insert: starting from the top level, go right until the element pointed is bigger than the one to search; then go back to the previous element and go down of one level and then, continue in that way. If the element is not found at bottom level, we need to insert it. To do it we have to rearrange the pointers like in the "normal list data structure".
After inserted, we have to compute the "*flip coin procedure*" to elevate the level of the node or

not. Flip a 50/50 coin, if *HEADS*, promote it to next level up and flip again until *TAIL* is reached; the number of HEADS determines the level of the node.

Having a list of $\log n$ levels (say $c \log n$) with high probability ($> 1 - \frac{1}{n^c}$) the complexity is $O(\log n)$.

## Describe how skip list delete works

The skip list is randomized list-like structure. It is a list with multiple levels and each node of *i* level has *i* pointers.

To delete a node with key *k*, we need to find it in the skip list and, once located, rearrange the pointers just like we do in "normal list".
To do so we start from the top level, go right until the element pointed is bigger than the one to search; then go back to the previous element and go down of one level and then, continue in that way until you find it; if the element is not found at level 1, it is not present in the skip list.
The delete operation works as for a normal linked list, but we must remember to delete it also from the upper level(s).
Having a list of $\log n$ levels (say $c \log n$) with high probability ($> 1 - \frac{1}{n^c}$) the complexity is $O(\log n)$.

## Describe what the memoization technique is and for what is relevant. List and discuss some examples using this technique.

Memoization is used to speed up computer programs by eliminating the repetitive computation of results, and by avoiding repeated calls to functions that process the same input.

We can use the memoization technique where the use of the previously calculated results comes into the picture. This kind of problem is mostly used in the context of recursion, especially with problems that involve overlapping subproblems.

There are two different approaches:

- Top-Down approach that uses Memoization
- Bottom-Up approach that uses Tabulation

For example, imagine calculating the factorial of 2, 3, 9 and 5.
We need to call the factorial 4 times, so we have O$(n * k)$ where *n* is representing how many numbers we have and *k* how many times we need to call the factorial method.
In this problem, we observe that, for example, the factorial of 2 is calculated 4 times or the factorial of 3, 3 times, so we can store the result of each individual factorial at the first calculation and then use that result when we need it ($O(1)$).

This is an example where memoization is useful.

## Describe the algorithm solving the Longest Common Subsequence problem

Given two sequences, find the length of longest subsequence present in both. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

The first approach is brute forcing, so trying to check all sub sequences of *x* and *y* but this is not efficient.

The LCS problem has an *optimal substructure*: the problem can be broken down into smaller, simpler subproblems, which can, in turn, be broken down into simpler subproblems, and so on, until, finally, the solution becomes trivial. It also has *overlapping subproblems*: the solutions to high-level subproblems often reuse solutions to lower level subproblems.

After computing a subproblem solution, we can store it into a table and in this way, we are able to reuse it avoiding redoing work. This technique is called *memoization*.

So, we need a matrix of dimension *d + 1* and initialize the first row and first column to *0*. Then, move to the new row and

$$if\ x[i] \neq y[i] \begin{cases} put\ the \max value\ between\ left\ cell\ and\ top\ cell \\ put\ the\ value\ of\ diagonal\ cell + 1\ otherwise \end{cases}$$

and so on.
In the end, we need do backward the matrix and we will find the LCS.

The complexity is $O(n * m)$ where *n* and *m* are respective size of the two sequences.

## Explain the difference between expected time and amortized time

*Expected time* is the time needed to do an operation on a data structure when we make some assumptions about the input and the state of the data structure.
*Amortized time* is the way to express the time complexity when an algorithm has the very bad time complexity only once in a while besides the time complexity that happens most of time.

Imagine having an array, each insertion costs $O(1)$. When it hits its capacity, we need to create a new one with doubled capacity and copy all the items from the old one to the new one which takes $O(n)$.

The insertion takes $O(n)$ when the capacity has been reached, and the *amortized time* for each insertion is $O(1)$.

## Describe for what the amortized analysis is meant and specifically discuss the accounting method

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyse a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.

**Accounting analysis**: we assign to every operation a fictitious cost *c'* instead of its real cost *c*; since we want an upper bound for every sequence of operations, for every *n* it must hold:

$$\sum_{i=1}^{n} c' \geq \sum_{i=1}^{n} c$$

For example, in a table, we can assign a cost of *3* to the insert operation. The real cost is *1*, which leaves us with $3 - 1 = 2$ in credit for every element.

We have that *dim(HT)* = *m* and, after we have done *m* insertion, we have a pool of

$$(3 - 1) * m = 2m$$

When the structure fills up, we must double the dimension of *T* (*2m*), copy the old values into the new one and add the *m* + *1*$^{th}$ value; after this we have that the pool is:

$$2m + 3 - 1 - m = m + 2$$

Now, adding *m – 1* elements we obtain a pool of:

$$m + 2 + (3 - 1) * (m - 1) = 3m$$

If we continue in this way, we can see that the credit balance never drops below *0* and the cost of *n* insertions is *Θ(n)*.

## Discuss how move to front heuristic works for self-organizing lists (asked twice)

Self-Organizing list is a list that re-organizes or re-arranges itself for better performance.

In a simple list, an item to be searched is looked for in a sequential manner which gives the time complexity of O(n) but not all the items are searched frequently.
So, a self-organizing list uses a property known as *locality of reference* that brings the most frequent used items at the head of the list. This increases the probability of finding the item at the start of the list and those elements which are rarely used are pushed to the back of the list.

In *move to front heuristic* (MTF), the re-organization of the list works a slightly different way because just the recently searched is moved to the front of the list.

It can be shown that an online algorithm using MTF is quite fast: it's linear in complexity with regard to an offline algorithm (one with perfect knowledge of the future requests like "God's algorithm"); in fact $T(MTF) \leq 4T(God's\ algorithm)$, so we say that MTF is *4-competitive*.

An online algorithm *A* is *α-competitive* if there exists a constant *k* such that for any sequence *S* of operations,

$$C_A(S) \leq \alpha * C_{OPT}(S) + k$$

where OPT is the optimal offline algorithm.

# EXTRA

## Amortized analysis – Potential analysis

The *potential method* is a method used to analyse the amortized time and space complexity of a data structure.

In the potential method, a function $\Phi$ is chosen that maps states of the data structure to non-negative numbers. If S is a state of the data structure, $\Phi$(S) represents work that has been accounted for ("*paid for*") in the amortized analysis but not yet performed. Thus, $\Phi$(S) may be thought of as calculating the amount of potential energy stored in that state.

Operation *i* transforms $D_{i-1}$ to $D_i$
The cost of *i* is $c_i$

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0 \; \forall \; i$
The amortized cost $\hat{c} = c_i + \Phi(D_i) - \Phi(D_{i-1})$

If $\Delta\Phi_i > 0$, then $\hat{c} > c_i$. Operation *i* stores work in the data structure for later use.

If $\Delta\Phi_i < 0$, then $\hat{c} < c_i$. The data structure delivers up stored work to help pay for operation *i*.

And since $\Phi(D_n) \geq 0 \text{ and } \Phi(D_0) = 0$ by definition:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \widehat{c_i}$$

That defines an upper bound on the total cost of the sequence of $n$ operations.

Consider the example of a table where

$$c_i = the\ cost\ of\ i^{th} insertion = \begin{cases} i \; if \; i-1 \; is \; an \; exact \; power \; of \; 2 \\ 1 \; otherwise \end{cases}$$

Define the potential of the table after the i[th] insertion: $\Phi(D_i) = 2i - 2^{\lceil \log i \rceil}$. ($2^{\lceil \log 0 \rceil} = 0$)
The amortized cost of the i[th] insertion is
$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \left(2i - 2^{\lceil \log i \rceil}\right) - \left(2(i-1) - 2^{\lceil \log(i-1) \rceil}\right) = 2 - 2^{\lceil \log i \rceil} + 2^{\lceil \log(i-1) \rceil}$

**Case 1**: i – 1 is an exact power of 2 →
$\hat{c}_i = i + 2 - 2^{\lceil \log i \rceil} + 2^{\lceil \log(i-1) \rceil} = i + 2 - 2(i-1) + (i-1) = i + 2 - 2i + 2 + i - 1 = 3$

**Case2**: i-1 is not an exact power of 2 →
$$\hat{c}_i = 1 + 2 - 2^{\lceil \log i \rceil} + 2^{\lceil \log(i-1) \rceil} = 3$$

Therefore, *n* insertions cost $\Theta(n)$ in the worst case.

## Amortized analysis – Aggregate analysis

In aggregate analysis we show that, for all sequences of *n* operations, there is an aggregate cost *T(n)* covering the worst-case cost for the sequence.

Consider a sequence of *n* insertions. The worst-case time to execute one insertion is *Θ(n)*.
Therefore, the worst-case time for *n* insertions is $n * \Theta(n) = \Theta(n^2)$.
Let $c_i = the\ cost\ of\ i^{th} insertion = \begin{cases} i \; if \; i-1 \; is \; an \; exact \; power \; of \; 2 \\ 1 \; otherwise \end{cases}$

(3[rd] insertions → $c_i$ = 3, 5[th] insertion → $c_i$ = 5, 9[th] → $c_i$ = 9)

So, cost of *n* insertions =

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log(n-1)} 2^j \leq 3n = \Theta(n)$$

Thus, the average cost of each dynamic table operations is $\frac{\Theta(n)}{n} = \Theta(1)$.

## Randomized algorithms – Las Vegas vs. Monte Carlo

Las Vegas and Monte Carlo are two different classes of randomized algorithms.

Las Vegas guarantees us the correct solution to our problem every run, instead, Monte Carlo may be wrong providing the solution.
In case of decision problems (answer is YES or NO), there are two kind of Monte Carlo algorithms:

1. one-sided error if sided error if the probability that it errs is zero for at least one of the possible outputs (YES/NO) that it produces.
2. Two-sided error sided error if there is a non-zero probability that it errs when it outputs either YES or NO.

# Disjoint sets

A disjoint set is a dynamic collection $\gamma = \{S_1, \dots S_k\}$ of disjoint sets. Each set $S_i$ is identified by one of its members, called representative. Each set pair is disjoint, so $S_i \cap S_j = \emptyset \ \forall i, j$.
Asking the representative of the same set twice without modifying it should return the same item.

The operations defined on a disjoint set are:

- *MAKE_SET (x)* creates a new set containing only one element $x$ which is also designed as its representative.
- *UNION (x, y)* creates a new set defined as the union of the 2 sets containing the input items $x$ and $y$, then it destroys the two sets that contain $x$ and $y$ before adding the newly created set to the collection in order to maintain the disjoint propriety.
- *FIND_SET (x)* given an element $x$ it returns the representative of the set containing $x$.

## List implementation

Such structure can be implemented as a list where each node of the list represents an element of a set and contains a pointer to the next element and a pointer to the first element to the list.
The first element of the list is designed as representative of the set.

Operations:

- *MAKE_SET (x)*, it creates a new list containing only *x*. The running time is constant.
- *FIND_SET (x)*, it follows the pointer of the first element of the list that is included in the node given as input. The running time is constant.
- *UNION (x, y)*, first it finds the representative of 2 sets containing the input elements, then one of the lists is appended to the end of the other and all pointers to the representative of the appended list are changed to make them point to the first element of the list they are appended to.

If we consider $n$ make-set operation followed by $n - 1$ merge set we have a total cost of $\Theta(n^2)$ in the worst-case (we always append the longest list to a list having one single element), therefore each operation of the sequence has amortized cost of $\Theta(n)$ which is not acceptable for most use cases.

## Forest implementation

Each set can be implemented as a rooted tree, the representative of the set is the root of the tree, and every element has a pointer to the parent node. The root points to itself (loop).

Operations:

- *MAKE_SET (x)* operation is implemented by creating a tree having one single node which is also the root of the tree. Executed in constant time.
- *UNION (x, y)* operation is implemented by letting the root of one of the input set point to the root of the other input set. Executed in constant time.
- *FIND_SET (x)* operation is implemented by following the pointer chain from the input node up to the root of the tree containing it. The cost of this operation depends on the height of the tree.

If we consider $n$ make-set operation followed by $n-1$ merge set we have a total cost of $\Theta(n^2)$ in the worst case (completely unbalanced tree), therefore each operation of the sequence has amortized cost of $\Theta(n)$. This can be mitigated by implementing *path compression* and *union-by-rank*.

- *UNION-BY-RANK* works by assigning to each node a rank, that is a number that represents an upper bound on the height of the node in the tree. When a union is performed, let the root with smaller rank point to the root with higher rank.
- *PATH COMPRESSION* is implemented as an optimization performed during the find-set operation and works by setting the parent node of all nodes encountered during the tree traversal equal to the root of the tree, reducing its height.

## Amdahl's Law vs Gustafson's Law

Amdahl proposed a model of computation that consists of characterizing the steps of the execution of a program as Serial or Parallelizable. Parallelizable segments can be executed in parallel whereas Serial segments cannot. In the proposed model, the execution of a program has a fixed fraction $f$ of parallelizable segments and the presence of serial segments limits the maximum speedup to a value strictly lower than the number of processors used for parallel execution. Since the parallelizable fraction of a program is fixed, we have (in this model):

$$SU_p = \frac{T_1}{T_p} = \frac{T_1}{T_1(1-f) + \frac{f * T_1}{p}} = \frac{1}{(1-f) + \frac{f}{p}} \quad \rightarrow_{p\to\infty} \frac{1}{1-f}$$

Given this relation it is possible to know the number of processors needed given the parallelizable fraction of a program. The relation is, however, pessimistic. For a program having $f = 0.9$ there is no point to have more than 10 processors regardless of the input size.

Gustafson pointed out how Amdahl's model is flawed since it relies on the assumption that the parallelizable portion of a program is fixed whereas this is not the case, and it is related to the input size. For many program the more data is to be analyzed, the more the program can be parallelized. The model proposed by Gustafson is a fixed time model. There is a portion of the total execution time called "serial time" $s$ that is the fraction of execution time that the program spends in serial execution. Under this assumption we have:

$$SU_p = \frac{T_1}{T_p} = \frac{s + p(1-s)}{s + (1-s)} = s + p(1-s)$$

The speedup linearly increases with the number of processors.

## Describe how insert, search and delete operations works on a treap

A treap is a randomized data structure that combines the properties of trees with the property of a heap; every element *e1* has a priority *p* and a key *k*. So, at any moment the tree is equivalent to a classic binary tree if the elements were inserted in the order of their priority.

Search: we want to search an element *x* in the treap.
It works like in a normal binary tree; we need to check, for every node, *key(x)* = *key(node)*, if it is we have found our element, otherwise we need to move left if *key(x)* < *key(node)* or right if *key(x)* > *key(node)*.

The complexity of the search operation is $O(\#elements\ on\ the\ search\ path)$.

Insert: we want to insert a new element *x* into our treap T.
First, choose a *prio(x)*, then add it as a leaf and restore the heap property in this way:

$$prio\big(parent(x)\big) > prio(x) \rightarrow \begin{cases} RotateRight\big(parent(x)\big)\ if\ x\ is\ left\ child \\ RotateLeft\big(parent(x)\big)\ otherwise \end{cases}$$

Delete: we want to remove a node *x* from the treap.
First, find it into the treap moving right or left depending on its priority, then:

- if *x* is a leaf, we can easily remove it;
- if *x* is not a leaf

$$find\ u\ (a\ child\ of\ x\ with\ smaller\ priority) \rightarrow \begin{cases} RotateRight(x)\ if\ u\ is\ left\ child \\ RotateLeft(x)\ otherwise \end{cases}$$

Then, delete *x*.

The complexity for insert and delete operation is $O(\log n)$ and the expected number of rotations is 2.

## PRAM structure and definition of speedup (partial exam 08/11/2022)

The PRAM (Parallel Random-Access Machine) is a machine model used to design parallel algorithms.
A PRAM is composed of infinitely many RAMs $M_i$ that share infinitely many input cells $X$, infinitely many output cells $Y$ and infinitely many shared memory cells used for communication and sharing of intermediate results. Each processor has an unbounded number of registers with unbounded length.
All machines composing the PRAM are identical and can recognize their indexes.
Based on the behavior of read and write operations on memory cells different PRAM models can be defined, namely

- Exclusive Read (ER): all processors can read from distinct memory cells simultaneously
- Exclusive Write (EW): all processors can write from distinct memory cells simultaneously
- Concurrent Read (CR): all processors can read from any memory cell simultaneously
- Concurrent Write (CW): all processors can read from any memory cell simultaneously

Any model that implements CW needs also coordination mechanisms regardless of the Read model used. The 3 solutions are

- Priority CW: allows to assign a priority to all processors, in case of a concurrent write on a given memory cell, the value that ends up to be written is the value written by the processor having highest priority. It is important to notice that in this case the processors are not undistinguishable.
- Common CW: all write operations are successful if and only if all processors that are trying to write a value in the memory cell are writing the same value.
- Random CW: the value that ends up to be written is randomly chosen between the ones that are being concurrently trying to be written.

The algorithms that are designed on PRAM are analysed based on the following definitions:

- $\boldsymbol{T^*(n)}$ : performance of the best sequential algorithm solving the same problem
- $\boldsymbol{T_p(n)}$ : performance of the algorithm running on p processors
- $\boldsymbol{T_\infty(n)}$**:** shortest possible running time given an unbounded number of processors

And the relevant quantities for analysis are:

- Speedup on p processors: $SU_p(n) \coloneqq \frac{T^*(n)}{T_p(n)}$
- Efficiency of the parallelization on p processors: $E_p(n) \coloneqq \frac{T_1(n)}{p*T_p(n)}$
- Cost: $C(n) \coloneqq p * T_p(n)$
- Work: $W(n) \coloneqq total\ number\ of\ operations$

## Describe the difference between online and offline algorithm

On-line algorithm A must execute the operation immediately without any knowledge of future operations (e.g., Tetris).
An off-line algorithm may see the whole sequence S in advance.